DELHI COLLEGE OF ENGINEERING

BAWANA ROAD, DELHI-42

LIBRARY

Class No. _____

Book No. _____

Accession No. _____

STUDY AND IMPLEMENTATION OF DYNAMIC SOURCE ROUTING PROTOCOL

A Project Report submitted in partial fulfillment of the requirements for the award of the Degree of

BACHELOR OF ENGINERING
IN
COMPUTER ENGINEERING

Project work done by

SHIV KUMAR VIKAS BANSAL ASHWANI KUMAR

Under the esteemed guidance of

Mr. RAJEEV KUMAR,lecturer
Department of Computer Engineering

Department of Computer Engineering
Delhi College of Engineering
DELHI UNIVERSITY
DELHI

CERTIFICATE

This is to certify that the project work entitled "**STUDY AND IMPLEMETATON OF DYNAMIC SOURCE ROUTING PROTOCOL**" Which is be submittted by **VIKAS BANSAL (2K1/COE/063),SHIV KUMAR(2K1/COE/053) ASHWANI KUMAR (2K1/COE/015)** to the Delhi College of Engineering in partial fulfillment of the degree of Bachelor of Engineering in **Computer engineering** is a record of bonafied poject work carried out by them They have worked under my guidance and have fulfilled the requirements for the submission of the project

Mr.RAJEEV KUMAR,Lecturer
Dept of Computer Engineering
Delhi College Of Engineering
Delhi

ACKNOWLEDGEMENTS

Behind every achievement of a student lies the unflinching effort of his teachers without whom as students we could never know the liveliness of hard work. And to this day and to the day we certainly feel it as our worldly pleasure to get living to this day of thanking them.

We would like to express our gratitude and sincere thanks to our project guide **Mr. Rajeev Kumar** lecturer, department of computer engineering, Delhi College of Engineering, for his esteemed guidance and incessant Support given in presenting this report successfully.

We would thanks our head of the department **Prof. D. R. Choudhary**, Ph.D., for his kind cooperation in bringing out this project work successful.

Finally we would like to thanks all staff members of the department for their constant encouragement and inspiration in bringing out this project report.

Abstract

The Dynamic Source Routing protocol (DSR) is a simple and efficient routing protocol designed specifically for use in multi-hop wireless ad hoc networks of mobile nodes. DSR allows the network to be completely self-organizing and self-configuring, without the need for any existing network infrastructure or administration. The protocol is composed of the two main mechanisms of *"Route Discovery"* and *"Route Maintenance"*, which work together to allow nodes to discover and maintain routes to arbitrary destinations in the ad hoc network. All aspects of the protocol operate entirely on-demand, allowing the routing packet overhead of DSR to scale automatically to only that needed to react to changes in the routes currently in use. The protocol allows multiple routes to any destination and allows each sender to select and control the routes used in routing its packets, for example for use in load balancing or for increased robustness. Other advantages of the DSR protocol include easily guaranteed loop-free routing, support for use in networks containing unidirectional links, use of only soft state in routing, and very rapid recovery when routes in the network change. The DSR protocol is designed mainly for mobile ad hoc networks of up to about two hundred nodes, and is designed to work well with even very high rates of mobility. This document specifies the operation of the DSR protocol for routing unicast IPv4 packets.

Contents

1. Introduction

The Dynamic Source Routing protocol (DSR) is a simple and efficient routing protocol designed specifically for use in multi-hop wireless ad hoc networks of mobile nodes. Using DSR, the network is completely self-organizing and self-configuring, requiring no existing network infrastructure or administration. Network nodes cooperate to forward packets for each other to allow communication over multiple *"hops"* between nodes not directly within wireless transmission range of one another. As nodes in the network move about or join or leave the network, and as wireless transmission conditions such as sources of interference change, all routing is automatically determined and maintained by the DSR routing protocol. Since the number or sequence of intermediate hops needed to reach any destination may change at any time, the resulting network topology may be quite rich and rapidly changing.

In designing DSR, we sought to create a routing protocol that had very low overhead yet was able to react very quickly to changes in the network. The DSR protocol provides highly reactive service in order to help ensure successful delivery of data packets in spite of node movement or other changes in network conditions.The DSR protocol is composed of two main mechanisms that work together to allow the discovery and maintenance of source routes in the ad hoc network:

- *Route Discovery* is the mechanism by which a node S wishing to send a packet to a destination node D obtains a source route to D. Route Discovery is used only when S attempts to send a packet to D and does not already know a route to D.

- *Route Maintenance* is the mechanism by which node S is able to detect, while using a source route to D, if the network topology has changed such that it can no longer use its route to D because a link along the route no longer works. When Route Maintenance

indicates a source route is broken, S can attempt to use any other route it happens to know to D, or can invoke Route Discovery again to find a new route for subsequent packets to D. Route Maintenance for this route is used only when S is actually sending packets to D.

In DSR, Route Discovery and Route Maintenance each operate entirely *"on demand"*. In particular, unlike other protocols, DSR requires no periodic packets of any kind at any layer within the network. For example, DSR does not use any periodic routing advertisement, link status sensing, or neighbor detection packets, and does not rely on these functions from any underlying protocols in the network. This entirely on-demand behavior and lack of periodic activity allows the number of overhead packets caused by DSR to scale all the way down to zero, when all nodes are approximately stationary with respect to each other and all routes needed for current communication have already been discovered. As nodes begin to move more or as communication patterns change, the routing packet overhead of DSR automatically scales to only that needed to track the routes currently in use. Network topology changes not affecting routes currently in use are ignored and do not cause reaction from the protocol.

All state maintained by DSR is *"soft state"* in that the loss of any state will not interfere with the correct operation of the protocol; all state is discovered as needed and can easily and quickly be rediscovered if needed after a failure without significant impact on the protocol. This use of only soft state allows the routing protocol to be very robust to problems such as dropped or delayed routing packets or node failures. In particular, a node in DSR that fails and reboots can easily rejoin the network immediately after rebooting; if the failed node was involved in forwarding packets for other nodes as an intermediate hop along one or more routes, it can also resume this forwarding quickly after rebooting, with no or minimal interruption to the routing protocol.

In response to a single Route Discovery (as well as through routing information from other packets overheard), a node may learn

and cache multiple routes to any destination. This support for multiple routes allows the reaction to routing changes to be much more rapid, since a node with multiple routes to a destination can try another cached route if the one it has been using should fail. This caching of multiple routes also avoids the overhead of needing to perform a new Route Discovery each time a route in use breaks. The sender of a packet selects and controls the route used for its own packets, which together with support for multiple routes also allows features such as load balancing to be defined. In addition, all routes used are easily guaranteed to be loop-free, since the sender can avoid duplicate hops in the routes selected.

The operation of both Route Discovery and Route Maintenance in DSR are designed to allow unidirectional links and asymmetric routes to be easily supported. In particular, in wireless networks, it is possible that a link between two nodes may not work equally well in both directions, due to differing antenna or propagation patterns or sources of interference. DSR allows such unidirectional links to be used when necessary, improving overall performance and network connectivity in the system.

This document specifies the operation of the DSR protocol for routing unicast IPv4 packets in multi-hop wireless ad hoc networks. The specification of DSR in this document provides a compatible base on which such features can be added, either independently or by integration with the DSR operation specified here.

2. Assumptions

The DSR protocol as described here is designed mainly for mobile ad hoc networks of up to about two hundred nodes, and is designed to work well with even very high rates of mobility. Other protocol features and enhancements that may allow DSR to scale to larger networks are outside the scope of this document. We assume in this document that all nodes wishing to communicate with other nodes within the ad hoc network are willing to participate fully in the protocols of the network. In particular, each node participating in the ad hoc network SHOULD also be willing to forward packets for other nodes in the network.

The diameter of an ad hoc network is the minimum number of hops necessary for a packet to reach from any node located at one extreme edge of the ad hoc network to another node located at the opposite extreme. We assume that this diameter will often be small (e.g., perhaps 5 or 10 hops), but may often be greater than 1. Packets may be lost or corrupted in transmission on the wireless network. We assume that a node receiving a corrupted packet can detect the error and discard the packet.

Nodes within the ad hoc network MAY move at any time without notice, and MAY even move continuously, but we assume that the speed with which nodes move is moderate with respect to the packet transmission latency and wireless transmission range of the particular underlying network hardware in use. In particular, DSR can support very rapid rates of arbitrary node mobility, but we assume that nodes do not continuously move so rapidly as to make the flooding of every individual data packet the only possible routing protocol.

A common feature of many network interfaces, including most current LAN hardware for broadcast media such as wireless, is the ability to operate the network interface in *"promiscuous"* receive mode. This mode causes the hardware to deliver every received packet to the network driver software without filtering based on link-

layer destination address. Although we do not require this facility, some of our optimizations can take advantage of its availability. Use of promiscuous mode does increase the software overhead on the CPU, but we believe that wireless network speeds are more the inherent limiting factor to performance in current and future systems; we also believe that portions of the protocol are suitable for implementation directly within a programmable network interface unit to avoid this overhead on the CPU. Use of promiscuous mode may also increase the power consumption of the network interface hardware, depending on the design of the receiver hardware, and in such cases, DSR can easily be used without the optimizations that depend on promiscuous receive mode, or can be programmed to only periodically switch the interface into promiscuous mode. Use of promiscuous receive mode is entirely optional.

Wireless communication ability between any pair of nodes may at times not work equally well in both directions, due for example to differing antenna or propagation patterns or sources of interference around the two nodes . That is, wireless communications between each pair of nodes will in many cases be able to operate bidirectionally, but at times the wireless link between two nodes may be only unidirectional, allowing one node to successfully send packets to the other while no communication is possible in the reverse direction. Although many routing protocols operate correctly only over bidirectional links, DSR can successfully discover and forward packets over paths that contain unidirectional links. Some MAC protocols, however, such as MACA , MACAW, or IEEE 802.11, limit unicast data packet transmission to bidirectional links, due to the required bidirectional exchange of RTS and CTS packets in these protocols and due to the link-layer acknowledgement feature in IEEE 802.11; when used on top of MAC protocols such as these, DSR can take advantage of additional optimizations, such as the ability to reverse a source route to obtain a route back to the origin of the original route.

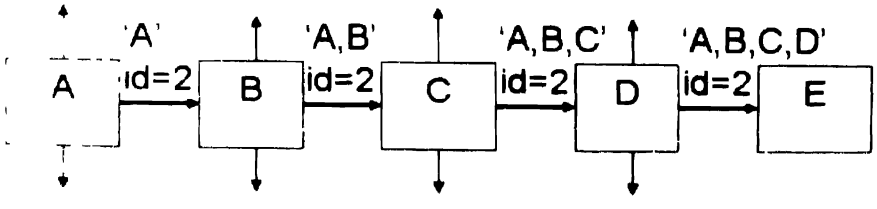
3. DSR Protocol Overview

This section provides an overview of the operation of the DSR protocol. The basic version of DSR uses explicit *"source routing"*, in which each data packet sent carries in its header the complete, ordered list of nodes through which the packet will pass. This use of explicit source routing allows the sender to select and control the routes used for its own packets, supports the use of multiple routes to any destination (for example, for load balancing), and allows a simple guarantee that the routes used are loop-free; by including this source route in the header of each data packet, other nodes forwarding or overhearing any of these packets can also easily cache this routing information for future use.

3.1. Basic DSR Route Discovery

When some source node originates a new packet addressed to some destination node, the source node places in the header of the packet a "*source route*" giving the sequence of hops that the packet is to follow on its way to the destination. Normally, the sender will obtain a suitable source route by searching its "*Route Cache*" of routes previously learned; if no route is found in its cache, it will initiate the Route Discovery protocol to dynamically find a new route to this destination node. In this case, we call the source node the "*initiator*" and the destination node the "*target*" of the Route Discovery.

For example, suppose a node A is attempting to discover a route to node E. The Route Discovery initiated by node A in this example would proceed as follows:



To initiate the Route Discovery, node A transmits a *"Route Request"* as a single local broadcast packet, which is received by (approximately) all nodes currently within wireless transmission range of A, including node B in this example. Each Route Request identifies the initiator and target of the Route Discovery, and also contains a unique request identification (2, in this example), determined by the initiator of the Request. Each Route Request also contains a record listing the address of each intermediate node through which this particular copy of the Route Request has been forwarded. This route record is initialized to an empty list by the initiator of the Route Discovery. In this example, the route record initially lists only node A.

When another node receives this Route Request (such as node B in this example), if it is the target of the Route Discovery, it returns a *"Route Reply"* to the initiator of the Route Discovery, giving a copy of the accumulated route record from the Route Request; when the initiator receives this Route Reply, it caches this route in its Route Cache for use in sending subsequent packets to this destination.

Otherwise, if this node receiving the Route Request has recently seen another Route Request message from this initiator bearing this same request identification and target address, or if this node's own address is already listed in the route record in the Route Request, this node discards the Request. Otherwise, this node appends its own address to the route record in the Route Request and propagates it by transmitting it as a local broadcast packet (with the same request identification). In this example, node B broadcast the Route Request, which is received by node C; nodes C and D each also, in turn, broadcast the Request, resulting in a copy of the Request being received by node E.

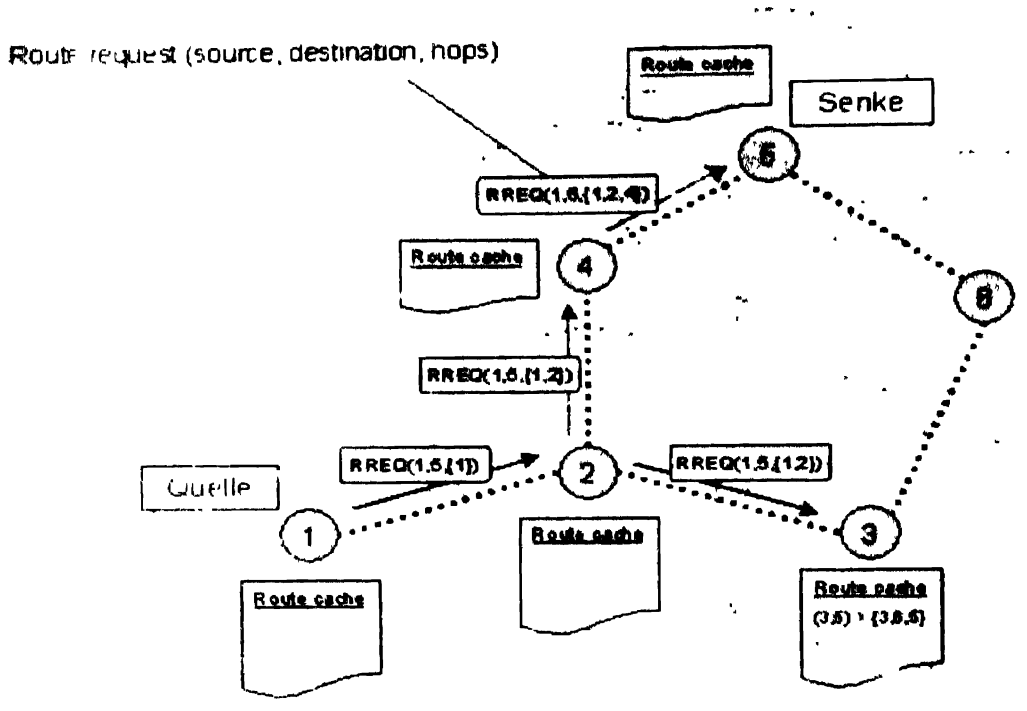


Fig: ROUTE REQUEST

In returning the Route Reply to the initiator of the Route Discovery, such as in this example, node E replying back to node A, node E will typically examine its own Route Cache for a route back to A, and if found, will use it for the source route for delivery of the packet containing the Route Reply. Otherwise, E SHOULD perform its own Route Discovery for target node A, but to avoid possible infinite recursion of Route Discoveries, it MUST piggyback this Route Reply on the packet containing its own Route Request for A. It is also possible to piggyback other small data packets, such as a TCP SYN packet on a Route Request using this same mechanism.

Node E could instead simply reverse the sequence of hops in the route record that it is trying to send in the Route Reply, and use this as the source route on the packet carrying the Route Reply itself. For MAC protocols such as IEEE 802.11 that require a bidirectional frame exchange as part of the MAC protocol, the discovered source route MUST be reversed in this way to return the Route Reply since it

tests the discovered route to ensure it is bidirectional before the Route Discovery initiator begins using the route; this route reversal also avoids the overhead of a possible second Route Discovery.

However, this route reversal technique will prevent the discovery of routes using unidirectional links, and in wireless environments where the use of unidirectional links is permitted, such routes may in some cases be more efficient than those with only bidirectional links, or they may be the only way to achieve connectivity to the target node.

When initiating a Route Discovery, the sending node saves a copy of the original packet (that triggered the Discovery) in a local buffer called the *"Send Buffer"*. The Send Buffer contains a copy of each packet that cannot be transmitted by this node because it does not yet have a source route to the packet's destination. Each packet in the Send Buffer is logically associated with the time that it was placed into the Send Buffer and is discarded after residing in the Send Buffer for some timeout period; if necessary for preventing the Send Buffer from overflowing, a FIFO or other replacement strategy MAY also be used to evict packets even before they expire.

While a packet remains in the Send Buffer, the node SHOULD occasionally initiate a new Route Discovery for the packet's destination address. However, the node MUST limit the rate at which such new Route Discoveries for the same address are initiated, since it is possible that the destination node is not currently reachable. In particular, due to the limited wireless transmission range and the movement of the nodes in the network, the network may at times become partitioned, meaning that there is currently no sequence of nodes through which a packet could be forwarded to reach the destination. Depending on the movement pattern and the density of nodes in the network, such network partitions may be rare or may be common.

If a new Route Discovery was initiated for each packet sent by a node in such a partitioned network, a large number of unproductive Route Request packets would be propagated throughout the subset of the ad hoc network reachable from this node. In order to reduce the

overhead from such Route Discoveries, a node SHOULD use an *exponential back-off algorithm* to limit the rate at which it initiates new Route Discoveries for the same target, doubling the timeout between each successive Discovery initiated for the same target. If the node attempts to send additional data packets to this is name destination node more frequently than this limit, the subsequent packets SHOULD be buffered in the Send Buffer until a Route Reply is received giving a route to this destination, but the node MUST NOT initiate a new Route Discovery until the minimum allowable interval between new Route Discoveries for this target has been reached. This limitation on the maximum rate of Route Discoveries for the same target is similar to the mechanism required by Internet nodes to limit the rate at which ARP Requests are sent for any single target IP address.

3.2. Basic DSR Route Maintenance

When originating or forwarding a packet using a source route, each node transmitting the packet is responsible for confirming that data can flow over the link from that node to the next hop. For example, in the situation shown below, node A has originated a packet for node E using a source route through intermediate nodes B, C, and D:

```
+-----+    +-----+    +-----+    +-----+    +-----+
|  A  |---->|  B  |---->|  C  |-->? |  D  |    |  E  |
+-----+    +-----+    +-----+    +-----+    +-----+
```

In this case, node A is responsible for the link from A to B, node is responsible for the link from B to C, node C is responsible for the link from C to D, node D is responsible for the link from D to E. An acknowledgement can provide confirmation that a link is capable of carrying data, and in wireless networks, acknowledgements are often provided at no cost, either as an existing standard part of the

MAC protocol in use (such as the link-layer acknowledgement frame defined by IEEE 802.11), or by a "*passive acknowledgement*" (in which, for example, B confirms receipt at C by overhearing C transmit the packet when forwarding it on to D).

If a built-in acknowledgement mechanism is not available, the node transmitting the packet can explicitly request a DSR-specific software acknowledgement be returned by the next node along the route; this software acknowledgement will normally be transmitted directly to the sending node, but if the link between these two nodes is unidirectional, this software acknowledgement could travel over a different, multi-hop path.

After an acknowledgement has been received from some neighbor, a node MAY choose to not require acknowledgements from that neighbor for a brief period of time, unless the network interface connecting a node to that neighbor always receives an acknowledgement in response to unicast traffic.

When a software acknowledgement is used, the acknowledgement request SHOULD be retransmitted up to a maximum number of times.

A retransmission of the acknowledgement request can be sent as a separate packet, piggybacked on a retransmission of the original data packet, or piggybacked on any packet with the same next-hop destination that does not also contain a software acknowledgement.

After the acknowledgement request has been retransmitted the maximum number of times, if no acknowledgement has been received, then the sender treats the link to this next-hop destination as currently "*broken*". It SHOULD remove this link from its Route Cache and SHOULD return a *"Route Error"* to each node that has sent a packet routed over that link since an acknowledgement was last received. For example, in the situation shown above, if C does not receive an acknowledgement from D after some number of requests, it would return a Route Error to A, as well as any other node that may have used the link from C to D since C last received an acknowledgement from D. Node A then removes this broken link from its cache; any retransmission of the original packet can be

performed by upper layer protocols such as TCP, if necessary. For sending such a retransmission or other packets to this same destination E, if A has in its Route Cache another route to E (for example, from additional Route Replies from its earlier Route Discovery, or from having overheard sufficient routing information from other packets), it can send the packet using the new route immediately. Otherwise, it SHOULD perform a new Route Discovery for this target.

3.3. Additional Route Discovery Features

3.3.1. Caching Overheard Routing Information

A node forwarding or otherwise overhearing any packet SHOULD add all usable routing information from that packet to its own Route Cache.

The usefulness of routing information in a packet depends on the directionality characteristics of the physical medium (Section 2), as well as the MAC protocol being used. Specifically, three distinct cases are possible:

- Links in the network frequently are capable of operating only unidirectionally (not bidirectionally), and the MAC protocol in use in the network is capable of transmitting unicast packets over unidirectional links.

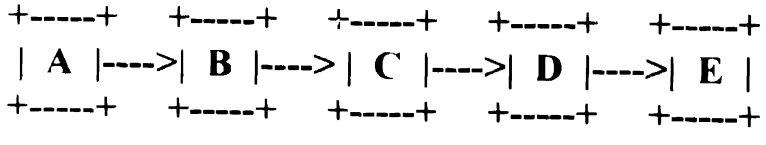
- Links in the network occasionally are capable of operating only unidirectionally (not bidirectionally), but this unidirectional restriction on any link is not persistent, almost all links are physically bidirectional, and the MAC protocol in use in the network is capable of transmitting unicast packets over unidirectional links.

- The MAC protocol in use in the network is not capable of transmitting unicast packets over unidirectional links; only

bidirectional links can be used by the MAC protocol for transmitting unicast packets. For example, the IEEE 802.11 Distributed Coordination Function (DCF) MAC protocol is capable of transmitting a unicast packet only over a bidirectional link, since the MAC protocol requires the return of a link-level acknowledgement packet from the receiver and also optionally requires the bidirectional exchange of an RTS and CTS packet between the transmitter and receiver nodes.

In the first case above, for example, the source route used in a data packet, the accumulated route record in a Route Request, or the route being returned in a Route Reply SHOULD all be cached by any node in the *"forward"* direction; any node SHOULD cache this information from any such packet received, whether the packet was addressed to this node, sent to a broadcast (or multicast) MAC address, or overheard while the node's network interface is in promiscuous mode. However, the *"reverse"* direction of the links identified in such packet headers SHOULD NOT be cached.

For example, in the situation shown below, node A is using a source route to communicate with node E:



As node C forwards a data packet along the route from A to E, it SHOULD add to its cache the presence of the *"forward"* direction links that it learns from the headers of these packets, from itself to D and from D to E. Node C SHOULD NOT, in this case, cache the *"reverse"* direction of the links identified in these packet headers, from itself back to B and from B to A, since these links might be unidirectional.

In the second case above, in which links may occasionally operate unidirectionally, the links described above SHOULD be cached in both directions. Furthermore, in this case, if node X

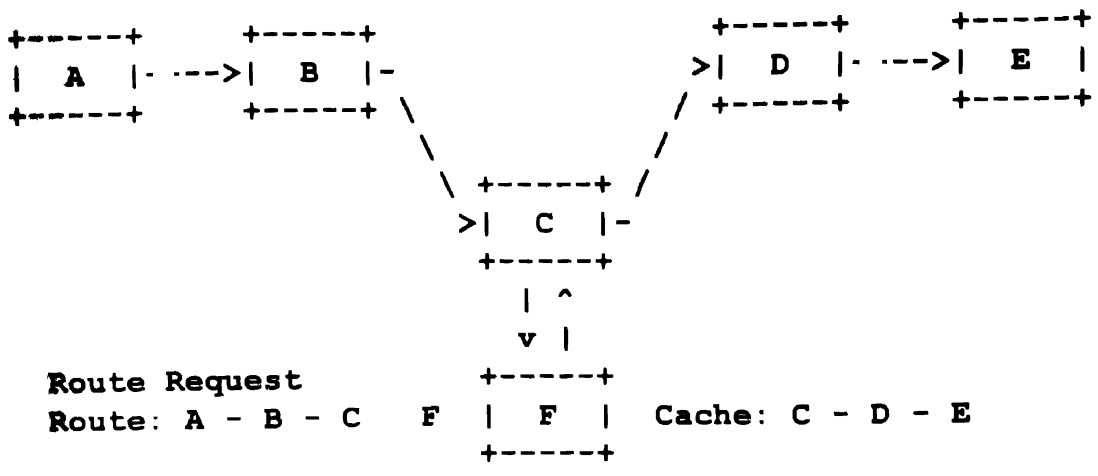
overhears (e.g., through promiscuous mode) a packet transmitted by node C that is using a source route from node A to E, node X SHOULD cache all of these links as well, also including the link from C to X over which it overheard the packet

In the final case, in which the MAC protocol requires physical bidirectionality for unicast operation, links from a source route SHOULD be cached in both directions, except when the packet also contains a Route Reply, in which case only the links already traversed in this source route SHOULD be cached, but the links not yet traversed in this route SHOULD NOT be cached.

3.3.2. Replying to Route Requests using Cached Routes

A node receiving a Route Request for which it is not the target, searches its own Route Cache for a route to the target of the Request. If found, the node generally returns a Route Reply to the initiator itself rather than forwarding the Route Request. In the Route Reply, this node sets the route record to list the sequence of hops over which this copy of the Route Request was forwarded to it, concatenated with the source route to this target obtained from its own Route Cache.

However, before transmitting a Route Reply packet that was generated using information from its Route Cache in this way, a node MUST verify that the resulting route being returned in the Route Reply, after this concatenation, contains no duplicate nodes listed in the route record. For example, the figure below illustrates a case in which a Route Request for target E has been received by node F, and node F already has in its Route Cache a route from itself to E:



The concatenation of the accumulated route record from the Route Request and the cached route from F's Route Cache would include a duplicate node in passing from C to F and back to C.

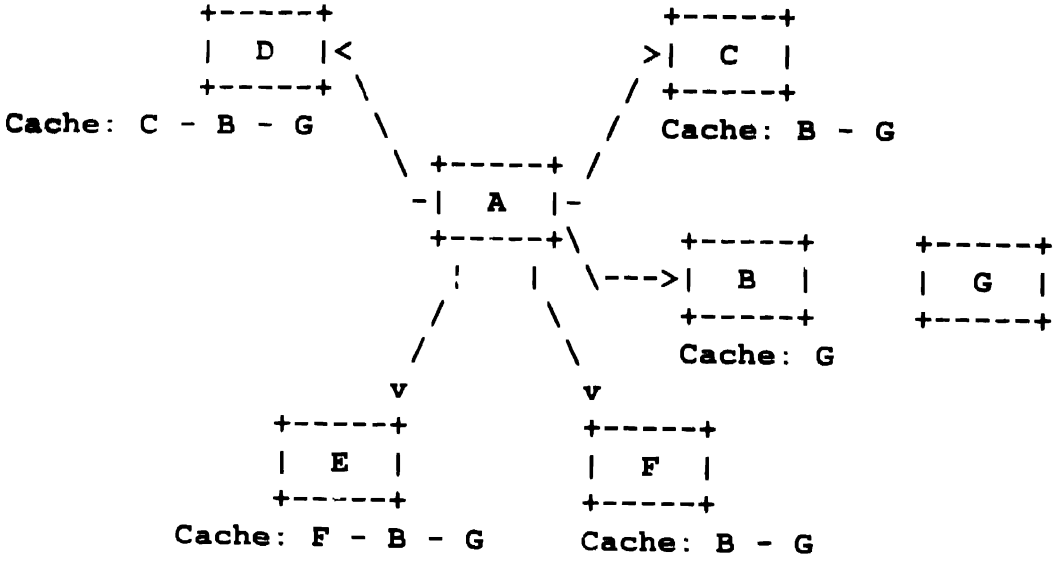
Node F in this case could attempt to edit the route to eliminate the duplication, resulting in a route from A to B to C to D and on to E, but in this case, node F would not be on the route that it returned in its own Route Reply. DSR Route Discovery prohibits node F from returning such a Route Reply from its cache; this prohibition increases the probability that the resulting route is valid, since node F in this case should have received a Route Error if the route had previously stopped working.

Furthermore, this prohibition means that a future Route Error traversing the route is very likely to pass through any node that sent the Route Reply for the route (including node F), which helps to ensure that stale data is removed from caches (such as at F) in a timely manner; otherwise, the next Route Discovery initiated by A might also be contaminated by a Route Reply from F containing the same stale route. If node F, due to this restriction on returning a Route Reply based on information from its Route Cache, does not return such a Route Reply, node F propagates the Route Request normally.

3.3.3. Preventing Route Reply Storms

The ability for nodes to reply to a Route Request based on information in their Route Caches, could result in a possible Route Reply "*storm*" in some cases. In particular, if a node broadcasts a Route Request for a target node for which the node's neighbors have a route in their Route Caches, each neighbor may attempt to send a Route Reply, thereby wasting bandwidth and possibly increasing the number of network collisions in the area.

For example, the figure below shows a situation in which nodes B, C, D, E, and F all receive A's Route Request for target G, and each has the indicated route cached for this target:



Normally, each of these nodes would attempt to reply from its own Route Cache, and they would thus all send their Route Replies at about the same time, since they all received the broadcast Route Request at about the same time. Such simultaneous Route Replies from different nodes all receiving the Route Request may cause local congestion in the wireless network and may create packet collisions among some or all of these Replies if the MAC protocol in use does not provide sufficient collision avoidance for these packets. In

addition, it will often be the case that the different replies will indicate routes of different lengths, as shown in this example.

In order to g node begins using a shorter route first. Specifically, this node MAY delay sending its own Route Reply for a random period

$$d = H * (h - 1 + r)$$

where h is the length in number of network hops for the route to be returned in this node's Route Reply, r is a random floating point number between 0 and 1, and H is a small constant delay (at least twice the maximum wireless link propagation delay) to be introduced per hop. This delay effectively randomizes the time at which each node sends its Route Reply, with all nodes sending Route Replies giving routes of length less than h sending their Replies before this node, and all nodes sending Route Replies giving routes of length greater than h sending their Replies after this node.

Within the delay period, this node promiscuously receives all packets, looking for data packets from the initiator of this Route Discovery destined for the target of the Discovery. If such a data packet received by this node during the delay period uses a source route of length less than or equal to h, this node may infer that the initiator of the Route Discovery has already received a Route Reply giving an equally good or better route. In this case, this node SHOULD cancel its delay timer and SHOULD NOT send its Route Reply for this Route Discovery.

3.3.4. Route Request Hop Limits

Each Route Request message contains a "*hop limit*" that may be used to limit the number of intermediate nodes allowed to forward that copy of the Route Request. This hop limit is implemented using the Time-to-Live (TTL) field in the IP header of the packet carrying the Route Request. As the Request is forwarded, this limit is decremented, and the Request packet is discarded if the limit reaches zero before finding the target. This Route Request hop limit can be

used to implement a variety of algorithms for controlling the spread of a Route Request during a Route Discovery attempt.

For example, a node MAY use this hop limit to implement a "*non-propagating*" Route Request as an initial phase of a Route Discovery. A node using this technique sends its first Route Request attempt for some target node using a hop limit of 1, such that any node receiving the initial transmission of the Route Request will not forward the Request to other nodes by re-broadcasting it. This form of Route Request is called a "*non-propagating*" Route Request; it provides an inexpensive method for determining if the target is currently a neighbor of the initiator or if a neighbor node has a route to the target cached (effectively using the neighbors' Route Caches as an extension of the initiator's own Route Cache). If no Route Reply is received after a short timeout, then the node sends a *"propagating"* Route Request (i.e., with no hop limit) for the target node.

As another example, a node MAY use this hop limit to implement an *"expanding ring"* search for the target. A node using this technique sends an initial non-propagating Route Request as described above; if no Route Reply is received for it, the node originates another Route Request with a hop limit of 2. For each Route Request originated, if no Route Reply is received for it, the node doubles the hop limit used on the previous attempt, to progressively explore for the target node without allowing the Route Request to propagate over the entire network. However, this expanding ring search approach could have the effect of increasing the average latency of Route Discovery, since multiple Discovery attempts and timeouts may be needed before discovering a route to the target node.

3.4. Additional Route Maintenance Features

3.4.1. Packet Salvaging

When an intermediate node forwarding a packet detects through Route Maintenance that the next hop along the route for that packet is

broken, if the node has another route to the packet's destination in its Route Cache, the node SHOULD *"salvage"* the packet rather than discarding it. To salvage a packet, the node replaces the original source route on the packet with the route from its Route Cache. The node then forwards the packet to the next node indicated along this source route. For example, in the situation shown in the example of Section 3.2, if node C has another route cached to node E, it can salvage the packet by replacing the original route in the packet with this new route from its own Route Cache, rather than discarding the packet.

When salvaging a packet, a count is maintained in the packet of the number of times that it has been salvaged, to prevent a single packet from being salvaged endlessly. Otherwise, it could be possible for the packet to enter a routing loop, as different nodes repeatedly salvage the packet and replace the source route on the packet with routes to each other.

As described in Section 3.2, an intermediate node, such as in this case, that detects through Route Maintenance that the next hop along the route for a packet that it is forwarding is broken, the node also SHOULD return a Route Error to the original sender of the packet, identifying the link over which the packet could not be forwarded. If the node sends this Route Error, it SHOULD originate the Route Error before salvaging the packet.

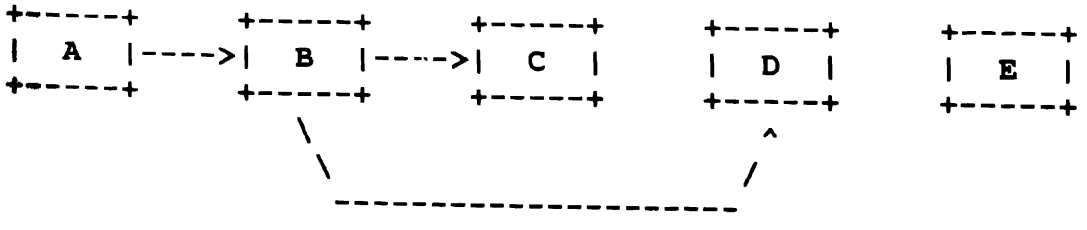
3.4.2. Queued Packets Destined over a Broken Link

When an intermediate node forwarding a packet detects through Route Maintenance that the next-hop link along the route for that packet is broken, in addition to handling that packet as defined for Route Maintenance, the node SHOULD also handle in a similar way any pending packets that it has queued that are destined over this new broken link. Specifically, the node SHOULD search its Network Interface Queue and Maintenance Buffer for packets for which the next-hop link is this new broken link. For each such packet currently queued at this node, the node SHOULD process that packet as follows:

- ❖ Remove the packet from the node's Network Interface Queue and Maintenance Buffer.
- ❖ Originate a Route Error for this packet to the original sender of the packet, as if the node had already reached the maximum number of retransmission attempts for that packet for Route Maintenance. However, in sending such Route Errors for queued packets in response to a single new broken link detected, the node SHOULD send no more than one Route Error to each original sender of any of these packets.
- ❖ If the node has another route to the packet's IP Destination Address in its Route Cache, the node SHOULD salvage the packet as described . Otherwise, the node SHOULD discard the packet.

3.4.3. Automatic Route Shortening

Source routes in use MAY be automatically shortened if one or more intermediate nodes in the route become no longer necessary. This mechanism of automatically shortening routes in use is somewhat similar to the use of passive acknowledgements. In particular, if a node is able to overhear a packet carrying a source route (e.g., by operating its network interface in promiscuous receive mode), then this node examines the unexpended portion of that source route. If this node is not the intended next-hop destination for the packet but is named in the later unexpended portion of the packet's source route, then it can infer that the intermediate nodes before itself in the source route are no longer needed in the route. For example, the figure below illustrates an example in which node D has overheard a data packet being transmitted from B to C, for later forwarding to D and to E:



In this case, this node (node D) SHOULD return a *"gratuitous"* Route Reply to the original sender of the packet (node A). The Route Reply gives the shorter route as the concatenation of the portion of the original source route up through the node that transmitted the overheard packet (node B), plus the suffix of the original source route beginning with the node returning the gratuitous Route Reply (node D). In this example, the route returned in the gratuitous Route Reply message sent from D to A gives the new route as the sequence of hops from A to B to D to E. When deciding whether to return a gratuitous Route Reply in this way, a node MAY factor in additional information beyond the fact that it was able to overhear the packet. For example, the node MAY decide to return the gratuitous Route Reply only when the overheard packet is received with a signal strenth or signal-to-noise ratio above some specific threshold. In addition, each node maintains a Gratuitous Route Reply Table, as described in Section 4.4, to limit the rate at which it originates gratuitous Route Replies for the same returned route.

3.4.4. Increased Spreading of Route Error Messages

When a source node receives a Route Error for a data packet that it originated, this source node propagates this Route Error to its neighbors by piggybacking it on its next Route Request. In this way, stale information in the caches of nodes around this source node will not generate Route Replies that contain the same invalid link for which this source node received the Route Error.

For example, in the situation shown in the example of Section 3.2, node A learns from the Route Error message from C, that the link from C to D is currently broken. It thus removes this link from its own Route Cache and initiates a new Route Discovery (if it has no other route to E in its Route Cache). On the Route Request packet initiating this Route Discovery, node A piggybacks a copy of this Route Error, ensuring that the Route Error spreads well to other nodes, and guaranteeing that any Route Reply that it receives (including those from other node's Route Caches) in response to this

Route Request does not contain a route that assumes the existence of this broken link.

4.1. Route Cache

All ad hoc network routing information needed by a node implementing DSR is stored in that node's Route Cache. Each node in the network maintains its own Route Cache. A node adds information to its Route Cache as it learns of new links between nodes in the ad hoc network; for example, a node may learn of new links when it receives a packet carrying a Route Request, Route Reply, or DSR source route.

Likewise, a node removes information from its Route Cache as it learns that existing links in the ad hoc network have broken; for example, a node may learn of a broken link when it receives a packet carrying a Route Error or through the link-layer retransmission mechanism reporting a failure in forwarding a packet to its next-hop destination.

Anytime a node adds new information to its Route Cache, the node SHOULD check each packet in its own Send Buffer (Section 4.2) to determine whether a route to that packet's IP Destination Address now exists in the node's Route Cache (including the information just added to the Cache). If so, the packet SHOULD then be sent using that route and removed from the Send Buffer.

It is possible to interface a DSR network with other networks, external to this DSR network. Such external networks may, for example, be the Internet, or may be other ad hoc networks routed with a routing protocol other than DSR. Such external networks may also be other DSR networks that are treated as external networks in order to improve scalability. The complete handling of such external networks is beyond the scope of this document. However, this document specifies a minimal set of requirements and features necessary to allow nodes only implementing this specification to interoperate correctly with nodes implementing interfaces to such external networks. This minimal set of requirements and features

involve the First Hop External (F) and Last Hop External (L) bits in a DSR Source Route option and a Route Reply option in a packet's DSR Options header. These requirements also include the addition of an External flag bit tagging each link in the Route Cache, copied from the First Hop External (F) and Last Hop External (L) bits in the DSR Source Route option or Route Reply option from which this link was learned. The Route Cache SHOULD support storing more than one route to each destination. In searching the Route Cache for a route to some destination node, the Route Cache is indexed by destination node address. The following properties describe this searching function on a Route Cache:

- Each implementation of DSR at any node MAY choose any appropriate strategy and algorithm for searching its Route Cache and selecting a ***"best"*** route to the destination from among those found. For example, a node MAY choose to select the shortest route to the destination (the shortest sequence of hops), or it MAY use an alternate metric to select the route from the Cache.

- However, if there are multiple cached routes to a destination, the selection of routes when searching the Route Cache MUST prefer routes that do not have the External flag set on any link. This preference will select routes that lead directly to the target node over routes that attempt to reach the target via any external networks connected to the DSR ad hoc network.

- In addition, any route selected when searching the Route Cache MUST NOT have the External bit set for any links other than possibly the first link, the last link, or both; the External bit MUST NOT be set for any intermediate hops in the route selected.

An implementation of a Route Cache MAY provide a fixed capacity for the cache, or the cache size MAY be variable. The following properties describe the management of available space within a node's Route Cache:

- Each implementation of DSR at each node MAY choose any appropriate policy for managing the entries in its Route Cache, such as when limited cache capacity requires a choice of which entries to retain in the Cache. For example, a node MAY chose a ***"least recently used"*** (LRU) cache replacement policy, in which the entry last used longest ago is discarded from the cache if a decision needs to be made to allow space in the cache for some new entry being added.

- However, the Route Cache replacement policy SHOULD allow outes to be categorized based upon ***"preference"***, where routes with a higher preferences are less likely to be removed from the cache. For example, a node could prefer routes for which it initiated a Route Discovery over routes that it learned as the result of promiscuous snooping on other packets. In particular, a node SHOULD prefer routes that it is presently using over those that it is not.

Any suitable data structure organization, consistent with this specification, MAY be used to implement the Route Cache in any node. For example, the following two types of organization are possible:

- In DSR, the route returned in each Route Reply that is received by the initiator of a Route Discovery (or that is learned from the header of overhead packets) represents a complete path (a sequence of links) leading to the destination node. By caching each of these paths separately, a ***"path cache"*** organization for the Route Cache can be formed.

A path cache is very simple to implement and easily guarantees that all routes are loop-free, since each individual route from a Route Reply or Route Request or used in a packet is loop-free. To search for a route in a path cache data structure, the sending node can simply search its Route Cache for any path (or prefix of a path) that leads to the intended destination node.

- Alternatively, a ***"link cache"*** organization could be used for the Route Cache, in which each individual link (hop) in the routes returned in Route Reply packets (or otherwise learned from the leader of overhead packets) is added to a unified graph data structure of this node's current view of the network topology. To search for a route in link cache, the sending node must use a more complex graph search algorithm, such as the well-known Dijkstra's shortest-path algorithm, to find the current best path through the graph to the destination node. Such an algorithm is more difficult to implement and may require significantly more CPU time to execute.

However, a link cache organization is more powerful than a path cache organization, in its ability to effectively utilize all of the potential information that a node might learn about the state of the network. In particular, links learned from different Route Discoveries or from the header of any overheard packets can be merged together to form new routes in the network, but this is not possible in a path cache due to the separation of each individual path in the cache.

The choice of data structure organization to use for the Route Cache in any DSR implementation is a local matter for each node and affects only performance; any reasonable choice of organization for the Route Cache does not affect either correctness or interoperability. Each entry in the Route Cache SHOULD have a timeout associated with it, to allow that entry to be deleted if not used within some time. The particular choice of algorithm and data structure used to implement the Route Cache SHOULD be considered in choosing the timeout for entries in the Route Cache. The configuration variable ***RouteCacheTimeout*** defined in specifies the timeout to be applied to entries in the Route Cache, although it is also possible to instead use an adaptive policy in choosing timeout values rather than using a single timeout setting for all entries; for example, the ***Link-MaxLife*** cache design (below) uses an adaptive timeout algorithm and does not use the ***RouteCacheTimeout*** configuration variable.

Link-MaxLife is an adaptive link cache in which each link in the cache has a timeout that is determined dynamically by the caching node according to its observed past behavior of the two nodes at the ends of the link; in addition, when selecting a route for a packet being sent to some destination, among cached routes of equal length (number of hops) to that destination, Link-MaxLife selects the route with the longest expected lifetime (highest minimum timeout of any link in the route). Use of the ***Link-MaxLife*** design for the Route Cache is recommended in implementations of DSR.

4.2. Route Request Table

The Route Request Table of a node implementing DSR records information about Route Requests that have been recently originated or forwarded by this node. The table is indexed by IP address. The Route Request Table on a node records the following information about nodes to which this node has initiated a Route Request:

- The Time-to-Live (TTL) field used in the IP header of the Route Request for the last Route Discovery initiated by this node for that target node. This value allows the node to implement a variety of algorithms for controlling the spread of its Route.
    - Request on each Route Discovery initiated for a target.
    - The time that this node last originated a Route Request for that target node.
    - The number of consecutive Route Discoveries initiated for this target since receiving a valid Route Reply giving a route to that target node.

The remaining amount of time before which this node MAY next attempt at a Route Discovery for that target node. When the node initiates a new Route Discovery for this target node, this field in the Route Request Table entry for that target node is initialized to the timeout for that Route Discovery, after which the node MAY initiate a new Discovery for that target. Until a valid Route Reply is received for this target node address, a node MUST implement a back-off algorithm in determining this timeout value for each successive Route Discovery initiated for this target using the same Time-to-Live (TTL) value in the IP header of the Route Request packet. The timeout between such consecutive Route Discovery initiations SHOULD increase by doubling the timeout value on each new initiation.

In addition, the Route Request Table on a node also records the following information about initiator nodes from which this node has received a Route Request:

- A FIFO cache of size ***RequestTableIds*** entries containing the identification value and target address from the most recent Route Requests received by this node from that initiator node.

Nodes SHOULD use an LRU policy to manage the entries in their Route Request Table. The number of Identification values to retain in each Route Request Table entry, ***RequestTableIds***, MUST NOT be unlimited, since, in the worst case, when a node crashes and reboots, the first ***RequestTableIds*** Route Discoveries it initiates after rebooting could appear to be duplicates to the other nodes in the network. In addition, a node SHOULD base its initial Identification value, used for Route Discoveries after rebooting, on a battery backed-up clock or other persistent memory device, in order to help avoid any possible such delay in successfully discovering new routes after rebooting; if no such source of initial Identification value is available, a node after rebooting SHOULD base its initial Identification value on a random number.

4.3. Gratuitous Route Reply Table

The Gratuitous Route Reply Table of a node implementing DSR records information about ***"gratuitous"*** Route Replies sent by this node as part of automatic route shortening. As described in Section 3.4.3, a node returns a gratuitous Route Reply when it overhears a packet transmitted by some node, for which the node overhearing the packet was not the intended next-hop node but was named later in the unexpended hops of the source route in that packet; the node overhearing the packet returns a gratuitous Route Reply to the original sender of the packet, listing the shorter route (not including the hops of the source route ***"skipped over"*** by this packet). A node uses its Gratuitous Route Reply Table to limit the rate at which it originates gratuitous Route Replies to the same original sender for the same node from which it overheard a packet to trigger the gratuitous Route Reply. Each entry in the Gratuitous Route Reply Table of a node contains the following fields:

- The address of the node to which this node originated a gratuitous Route Reply.

- The address of the node from which this node overheard the packet triggering that gratuitous Route Reply.

- The remaining time before which this entry in the Gratuitous Route Reply Table expires and SHOULD be deleted by the node

When a node creates a new entry in its Gratuitous Route Reply Table, the timeout value for that entry should be initialized to the value ***GratReplyHoldoff.*** When a node overhears a packet that would trigger a gratuitous Route Reply, if a corresponding entry already exists in the node's Gratuitous Route Reply Table, then the node SHOULD NOT send a gratuitous Route Reply for that packet. Otherwise (no corresponding entry already exists), the node

SHOULD create a new entry in its Gratuitous Route Reply Table to record that gratuitous Route Reply, with a timeout value of ***GratReplyHoldoff.***

4.4. Blacklist

When a node using the DSR protocol is connected through an interface that requires physically bidirectional links for unicast transmission, it MUST maintain a blacklist. A Blacklist is a table, indexed by neighbor address, that indicates that the link between this node and the specified neighbor may not be bidirectional. A node places another node's address in this list when it believes that broadcast packets from that other node reach this node, but thatunicast transmission between the two nodes is not possible. For example, if a node forwarding a Route Reply discovers that the next hop is unreachable, it places that next hop in the node's blacklist.

Once a node discovers that it can communicate ***bidirectionally*** with one of the nodes listed in the blacklist, it SHOULD remove that node from the blacklist. For example, if node A has node B in its blacklist, but A hears B forward a Route Request with a hop list indicating that the broadcast from A to B was successful, then A SHOULD remove B from its blacklist.

A node MUST associate a state with each node in the blacklist, specifying whether the unidirectionality is ***"questionable"*** or ***"probable"***. Each time the unreachability is positively determined, the node SHOULD set the state to ***"probable"***. After the unreachability has not been positively determined for some amount of time, the state should revert to ***"questionable"***. A node MAY expire nodes from its blacklist after a reasonable amount of time.

If sending the Route Reply to the initiator of the Route Request does not require performing a Route Discovery, a node SHOULD send a unicast Route Reply in response to every Route Request it receives for which it is the target node.

BroadcastJitter	10 milliseconds
RouteCacheTimeout	300 seconds
SendBufferTimeout	30 seconds
RequestTableSize	64 nodes
RequestTableIds	16 identifiers
MaxRequestRexmt	16 retransmissions
MaxRequestPeriod	10 seconds
RequestPeriod	500 milliseconds
NonpropRequestTimeout	30 milliseconds
RexmtBufferSize	50 packets
MaintHoldoffTime	250 milliseconds
MaxMaintRexmt	2 retransmissions
TryPassiveAcks	1 attempt
PassiveAckTimeout	100 milliseconds
GratReplyHoldoff	1 second

In addition, the following protocol constant MUST be supported by any implementation of the DSR protocol:

MAX_SALVAGE_COUNT	15 salvages

5. IANA Considerations

This document specifies the DSR Options header, which requires an IP Protocol number. This document also specifies the DSR Flow State header, which requires an IP Protocol number.

In addition, this document proposes use of the value ***"No Next Header"*** (originally defined for use in IPv6) within an IPv4 packet, to indicate that no further header follows a DSR Options header.

Finally, this document introduces a number of DSR options for use in the DSR Options header, and additional new DSR options may be defined in the future. Each of these options requires a unique Option Type value, with the most significant 3 bits (that is, Option Type & 0xE0)
encoded . It is necessary only that each Option Type value be unique, that they be unique in the remaining 5 bits of the value after these 3 most significant bits.

6. Security Considerations

This document does not specifically address security concerns. This document does assume that all nodes participating in the DSR protocol do so in good faith and without malicious intent to corrupt the routing ability of the network.

Depending on the threat model, a number of different mechanisms can be used to secure DSR. For example, in an environment where node compromise is unrealistic and where where all the nodes participating in the DSR protocol share a common goal that motivates their participation in the protocol, the communications between the nodes can be encrypted at the physical channel or link layer to prevent attack by outsiders. Cryptographic approaches can resist stronger attacks.

Appendix A. Link-MaxLife Cache Description

As guidance to implementors of DSR, the description below outlines the operation of a possible implementation of a Route Cache for DSR that has been shown to outperform other other caches studied in detailed simulations. Use of this design for the Route Cache is recommended in implementations of DSR.

This cache, called ***"Link-MaxLife"*** is a link cache, in that each individual link (hop) in the routes returned in Route Reply packets (or otherwise learned from the header of overhead packets) is added to a unified graph data structure of this node's current view of the network topology. To search for a route in this cache to some destination node, the sending node uses a graph search algorithm, such as the well-known Dijkstra's shortest-path algorithm, to find the current best path through the graph to the destination node.

The Link-MaxLife form of link cache is adaptive in that each link in the cache has a timeout that is determined dynamically by the caching node according to its observed past behavior of the two nodes at the ends of the link; in addition, when selecting a route for a packet being sent to some destination, among cached routes of equal length (number of hops) to that destination, Link-MaxLife selects the route with the longest expected lifetime (highest minimum timeout of any link in the route).

Specifically, in Link-MaxLife, a link's timeout in the Route Cache is chosen according to a ***"Stability Table"*** maintained by the caching node. Each entry in a node's Stability Table records the address of another node and a factor representing the perceived ***"stability"*** of this node. The stability of each other node in a node's Stability Table is initialized to InitStability. When a link from the Route Cache is used in routing a packet originated or salvaged by that node, the stability metric for each of the two endpoint nodes of that link is incremented by the amount of time since that link was last used, multiplied by ***StabilityIncrFactor (StabilityIncrFactor >= 1);*** then a link is observed to break and the link is thus removed from the

Route Cache, the stability metric for each of the two endpoint nodes of that link is multiplied by ***StabilityDecrFactor (StabilityDecrFactor < 1).***

When a node adds a new link to its Route Cache, the node assigns a lifetime for that link in the Cache equal to the stability of the less "**stable**" of the two endpoint nodes for the link, except that a link is not allowed to be given a lifetime less than ***MinLifetime.*** When a link is used in a route chosen for a packet originated or salvaged by this node, the link's lifetime is set to be at least UseExtends into the future; if the lifetime of that link in the

Route Cache is already further into the future, the lifetime remains unchanged. When a node using ***Link-MaxLife*** selects a route from its Route Cache for a packet being originated or salvaged by this node, it selects the shortest-length route that has the longest expected lifetime (highest minimum timeout of any link in the route), as opposed to simply selecting an arbitrary route of shortest length.

The following configuration variables are used in the description of ***Link-MaxLife*** above. The specific variable names are used for demonstration purposes only, and an implementation is not required to use these names for these configuration variables. For each configuration variable below, the default value is specified to simplify configuration. In particular, the default values given below are chosen for a DSR network where nodes move at relative velocities between 12 and 25 seconds per transmission radius.

InitStability	25 seconds
StabilityIncrFactor	4
StabilityDecrFactor	1/2
MinLifetime	1 second
UseExtends	120 seconds

A Distributed Adaptive Cache Update Algorithm for the Dynamic Source Routing Protocol-:

On-demand routing protocols use route caches to make routing decisions. Due to frequent topology changes, cached routes easily become stale. To address the cache staleness issue in DSR (the Dynamic Source Routing protocol), prior work mainly used heuristics with ad hoc parameters to predict the lifetime of a link or a route. However, heuristics cannot accurately predict timeouts because topology changes are unpredictable.

We define a new cache structure called a cache table to maintain the information necessary for cache updates. When a node detects a link failure, our algorithm proactively notifies all reachable nodes that have cached the broken link in a distributed manner. We compare our algorithm with DSR with path caches and with Link-MaxLife through detailed simulations. We show that our algorithm significantly outperforms DSR with path caches and with Link-MaxLife.

Introduction-:

In mobile ad hoc networks, nodes move arbitrarily, cooperating to forward packets to enable communication between nodes not within wireless transmission range. Frequent topology changes present the fundamental challenge to routing protocols.

Routing protocols for ad hoc networks can be classified into two main types: proactive and reactive (on-demand). Proactive protocols attempt to maintain up-to-date routing information to all nodes by periodically disseminating topology updates throughout the network. On-demand protocols attempt to discover a route to a destination only when a node originates a packet.

On-demand routing protocols use route caches to avoid the overhead and the latency of initiating a route discovery for each data packet. However, due to mobility, cached routes easily become stale. Itincreases packet delivery latency due to expensive link failure detections, and increases routing overhead. When responding to route requests from caches

is used, stale routes will be quickly propagated to other nodes, aggravating the situation. Stale routes also seriously degrade TCP performance. To address the cache staleness issue, prior work proposed to use adaptive timeout mechanisms. Such mechanisms use heuristics with ad hoc parameters to predict the timeout of a link or a route. However, predetermined choice of ad hoc parameters for certain scenarios may not work well for others. The effectiveness of heuristics is limited by unpredictable topology changes. If timeout is set too short, valid links or routes will be removed; subsequent route discoveries introduce significant overhead. If timeout is set too long, stale routes will stay in caches. In addition, DSR uses a small cache size. Small caches with FIFO help evict stale routes but also remove valid ones.

DSR consists of two on-demand mechanisms: Route Discovery and Route Maintenance. When a source wants to send packets to a destination to which it does not have a route, it initiates a Route Discovery by broadcasting a ROUTE REQUEST. A node receiving a ROUTE REQUEST checks whether it has a route to the destination in its cache. If it has, it sends a ROUTE REPLY to the source including a source route, the concatenation of the source route in the ROUTE REQUEST and the cached route. Otherwise, it adds its address to the source route in the packet and rebroadcasts the ROUTE REQUEST. When the ROUTE REQUEST reaches the destination, the destination sends a ROUTE REPLY containing the source route to the source. When forwarding a ROUTE REPLY, a node stores the route starting from itself to the destination. Upon receiving the ROUTE REPLY, the source caches the source route.

In Route Maintenance, a node forwarding a packet is responsible for confirming that the packet has been received by the next hop. If no acknowledgement is received after the maximum number of retransmissions, this node will send a ROUTE ERROR to the source, indicating the broken link. Each node receiving a ROUTE ERROR removes from its cache the routes containing the broken link.

Besides Route Maintenance, DSR uses two mechanisms to remove stale routes. First, a source node piggybacks the last known broken link information on the next ROUTE REQUEST (called GRATUITOUS ROUTE ERROR) to clean more nodes. Second, it relies on heuristics: a

small cache size with FIFO replacement policy for path caches, and adaptive timeout mechanisms for link caches, where the timeout of a link is predicted based on observed link usages and breakages.

3 The Distributed Cache Update Algorithm-:

3.1 The Impact of Stale Routes-:

1.Causing packet losses, increasing packet delivery latency and routing overhead. These effects will become more significant as mobility, traffic load, or network size increases, because more routes will become stale and/or stale routes will affect more traffic sources.

2.Degrading TCP performance. Since TCP cannot distinguish between packet losses caused by route failures from those caused by congestion, it will falsely invoke congestion control mechanisms, resulting in the reduction in throughput.

3 Wasting the energy of source nodes and intermediate nodes. If stale routes are not removed quickly, TCP will retransmit lost packets still using stale routes.

Cache Update algorithm-:

Fast cache updating is important for reducing the adverse effects of stale routes. It is also necessary to constrain cache update notifications to the nodes that have cached a broken link in order to avoid the overhead of notifying other nodes. Thus, our goal is this: when a node detects a link failure, all reachable nodes whose caches contain the broken link will be notified about the link failure.

To achieve this goal, we make use of the information obtained from route discoveries and data transmission. We define a cache table to gather and maintain the information necessary for cache updates. Each node maintains two types of information for each route in its cache table: how well the routing information is synchronized among nodes on a route, and

which neighbor node outside a route has learned which link of the route. By keeping such local information, a node knows which neighbor node needs to be notified about a broken link. A node receiving a cache update notification uses the local information kept in its cache table to determine and notify the neighbor nodes that have cached the broken link. Thus, a broken link information will be quickly propagated to all reachable nodes whose caches contain that link.

3.3 The Definition of a Cache Table-:

A cache table has no capacity limit and thus its size changes as needed. Each entry of a cache table contains four fields:

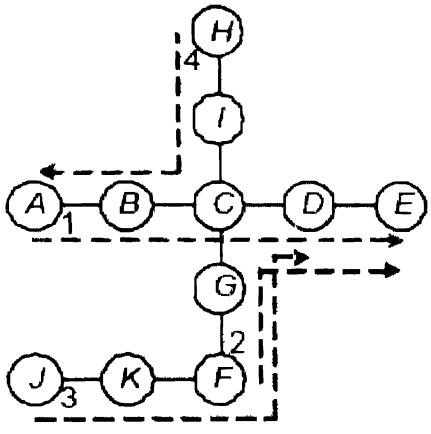


Figure 1: An Example of a Network with Four Flows

a.)Route: It is a route a node learns. A node first stores the links from itself to a destination from a ROUTE REPLY and later completes the sub-route stored before by adding upstream links from the first data packet. If no route in the table is a sub-route of the source route, it stores the complete source route from the data packet.

b.)SourceDestination: It is the source and destination pair.
c.)DataPackets: It records whether a node has forwarded 0, 1, or 2 data packets using the route. This field indicates to what extent the routing

information is synchronized among nodes on the route. It is 0 when the node stores downstream links from a ROUTE REPLY; it is incremented to ' when the node forwards the first data packet; and it is incremented to 2 when the node forwards the second data packet.

d.)ReplyRecord: When the node informs a neighbor of a sub-route through a ROUTE REPLY, it records the neighbor and the links used as an entry. If some entry contains a broken link, a node knows which neighbor it needs to notify about the link failure. This field has no capacity limit and thus its size changes as needed.

Information Collection and Maintenance for Cache Updates-:

During route discoveries and data transmission, we use two algorithms to collect and maintain the information necessary for cache updates. addRoute and findRoute.

We use a network shown in Fig. 1 in our examples. We will use S-D for SourceDestination, dP for DataPackets, and replyRec for ReplyRecord in the headers of tables describing the content of caches. Initially, there are no flows and all nodes' caches are empty. Node A initiates a route discovery to E and a ROUTE REPLY is sent from E to A. Upon receiving a ROUTE REPLY, each
intermediate node creates a new entry in its cache (addRoute: 6 11 in the Appendix). For instance, node C creates an entry consisting of four fields: (1) a route containing downstream links: CDE; (2) the source and destination pair: AE; (3) the number of data packets it received from the source node A: 0;
(4) which neighbor will learn which route: B will learn CDE. This is described as:

Route	S-D	dP	replyRec
CDE	AE	0	B — CDE

When A receives the ROUTE REPLY, it adds to its cache.

A :

Route	S-D	dP
ABCDE	*A E*	0

When node A uses this route to send the first data packet, its entry is updated as (findRoute):

A

Route	S-D	dP
ABCDE	*A E*	1

Each node receiving the data packet updates its cache entry. For instance, node C increments DataPackets to 1, replaces CDE by ABCDE (addRoute), and removes the entry in the ReplyRecord field (addRoute) because the complete route indicates A and B have cached all links of the route.

Thus, C's cache is:

C

Route	S-D	dP
ABCDE	*A E*	1

When E receives the first data packet, it creates a new entry (addRoute: 19) and its cache is the same as that of C. When C receives the second data packet, it increments dP to 2 (addRoute). Now, assume that C receives a ROUTE REQUEST from G with source F and destination D. Before sending a ROUTE REPLY to G, C will extend its cacheEntry to (findRoute).

C

Route	S-D	dP	replyRec
ABCDE	*A E*	2	*G← CDE*

Node G creates a cacheEntry(addRoute) before sending a ROUTE REPLY to F:

	Route	S-D	dP	replyRec
G	GFE	F E	0	F – GFE

When F gets the ROUTE REPLY, it inserts into its cache:

	Route	S-D	dP
F	FGFE	F E	0

Now, assume C receives a ROUTE REQUEST from I with source H and destination A. C extends its cache entry to (find-Route):

Route	S-D	dP	replyRec	replyRec
ABCDE	A E	1	C – CDE	A – CBA

If a node does not cache a source route and no sub-route can be completed, it creates a new entry and add the source route to its cache (addRoute). since the node knows that all the upstream nodes have stored the downstream links. For example, assume flow 2 starts. When it reaches D, D inserts the second entry into its cache:

Whenever a route is completed or a new source route is added, a node always checks the ReplyRecord to see whether the concatenation of two fields in an entry is a sub-route of the source route (addRoute:). If so, it removes that entry. Later, assume that after transmitting at least two packets, F receives a ROUTE REQUEST from K with source J and destination D. Before transmitting ROUTE REPLY to K, F extends its cache entry:

Each of J and K will save in its cache the route from itself to D.

The Distributed Adaptive Cache Update Algorithm-:

In this section, we first show several examples for the cache update algorithm.

Figure 2 Scenario 1

Examples-:

Scenario 1 Here we focus on DataPackets and ReplyRecord in a simple case, as shown in Fig. 2. Assume that A has initiated a route discovery to E. Before any data packet from flow 1 with route r1 = ABCDE reaches C, C discovers that the link CD is broken. In its cache it finds:

C.

Route	S-D	dP	replyRec
CDE	AE	0	B – CDE

Since DataPackets is 0, C knows that CDE is a pre-active route, and therefore no downstream nodes need to be notified, since they did not cache the broken link when forwarding a ROUTE REPLY. Node C needs to check whether some neighbor nodes have cached the broken link (cacheUpdate:). It notifies its neighbor B and removes the entry from its cache. B notifies A and cleans its cache. For another case, assume that A started transmissions for flow 1. While attempting to transmit a data packet, C detects that CD is broken. C's cache is:

C.

Route	S-D	dP
ABCDE	AE	3

Since this packet is a data packet, d is 1 or 2. Thus, upstream nodes need to be notified about the broken link. C adds its upstream neighbor B to a list consisting of nodes to be notified (cacheUpdate: 7- 10). If d 1 and the route being examined is the same as the source route in the packet, which means that the packet is the first data packet, then no downstream node needs to be notified. If d 2, then the packet is at least the second data packet arriving at C. (If d = 1 and the route being examined is different from the source route in the packet, then the route being examined has been synchronized by its first data packet. We handle this case the same as the case where d = 2.) There fore, at least one data packet has reached D and E and thus they have cached the route ABCDE. C searches in its cache for a shortest route to reach one of the downstream nodes (cacheUpdate:11 -18). Assume that it finds a route to E. So C notifies E. As the table entry does not contain any ReplyRecord, no neighbor has learned a sub-route from C.

C then removes the table entry. E in turn notifies D (cacheUpdate: 39–40). If no route to either D or E is found, they will not be notified at this time.

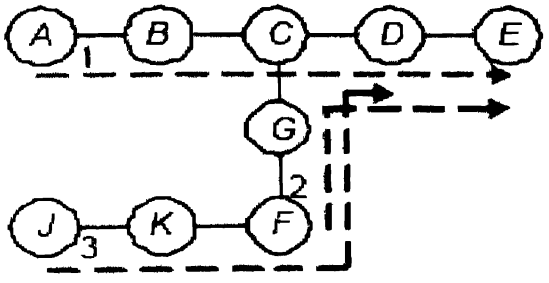


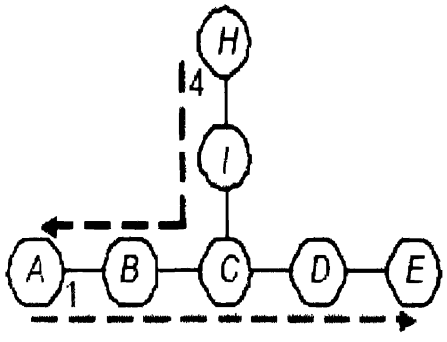
Figure 3: Scenario 2

Scenario 2 Here we focus on how ReplyRecord are handled in a slightly complex case, as shown in Fig. 3. As before, first A discovers r1 = ABCDE, then F discovers r2 = FGCDE for flow 2 after receiving the ROUTE REPLY sent by C. Finally, J discovers r3 = JFKD for flow 3 after receiving the ROUTE REPLY sent by F. r1 is an active route and both r2 and r3 are pre-active routes. When transmitting packets for flow 1, C discovers that CD is broken.

	Route	S-D	dP	replyRec
C	*ABCDE*	*AE*	2	*G* ← *CDE*
G	*GCDE*	*FE*	0	*F* ← *GCDE*
F	*FGCDE*	*FE*	0	*K* ← *FGCD*
K	*KFGCD*	*JD*	0	*J* ← *KFGCD*
J	*JKFGCD*	*JD*	0	

C handles the upstream and the downstream nodes of r1 the same as the second case in scenario 1. It also finds that it has notified a neighbor G of a route containing CD. So C notifies G that CD is broken and deletes r1 from its cache (cacheUpdate: 44–48). After G examines its cache, it notifies F

and cleans its cache. F notifies K and cleans its cache. K notifies J and cleans its cache. Finally, J cleans its cache.



Scenario 3 Here we focus on how to handle a broken link through which two (or more) flows in opposite directions are flowing, as shown in Fig. 4. Assume that only flows 1 and 4 have started. Here r1 =ABCDE and r4 = HICBA. While transmitting a packet for flow 4, C detects that CB is broken.

Scenario 3 Here we focus on how to handle a broken link through which two (or more) flows in opposite directions are flowing, as shown in Fig. 4. Assume that only flows 1 and 4 have started. Here r1 =ABCDE and r4 = HICBA. While transmitting a packet for flow 4, C detects that CB is broken.

The cache of C is:

	Route	S-D	dP
C.	ABCDE	AE	2
C.	HICBA	HA	2

Nodes that need to be notified about the broken link are: H, I (upstream in r4); A, B (upstream in r1 and downstream in r4); and D, E (downstream in r1). Node C attempts to find a shortest path to reach A or B; it also notifies D and I about the broken link (cacheUpdate:). A node receiving a notification determines a list of neighbor nodes it is responsible for notifying. For example, node I knows that it needs to notify its upstream node H (cacheUpdate:), and D knows that it needs to notify its downstream node, here E (cacheUpdate:).

Concise Description-:

A node learns a broken link either by detecting it itself or through a cache update notification from another node. In either case, the cache update algorithm is started reactively to examine each entry of the cache table. For each route containing a broken link, the algorithm determines a set of neighborhood nodes it needs to notify about the broken link, its upstream and/or downstream neighbors as well as other neighbors outside the route. Finally, the algorithm produces a notification list, which includes nodes to be notified about the broken link and routes to reach those nodes.

A node sends cache update notifications through ROUTE ERRORS. In each route containing a broken link, the link is in the forward direction if the flow using that route crosses the link in the same direction

as the flow that has detected the breakage; otherwise it is in the backward direction. For these two types of links, the operation of the algorithm is symmetric. When a node detects a link failure, it examines each entry of its cache. If a route contains the broken link in the forward direction (cacheUpdate)then the algorithm does the following steps:

a.) If dataPackets is 2 or 1, then upstream nodes (if any) need to be notified, but only the upstream neighbor is added to the notification list.

b.) If DataPackets is 2, or 1 and the route being examined is different from the source route in the packet, then downstream nodes need to be notified. The algorithm tries to find a shortest path to reach one of the downstream nodes. For both cases, the first data packet has traversed that route and thus all downstream nodes have cached the
broken link.

c.) If DataPackets is 0, there is no upstream node and no downstream nodes need to be notified.

Some Implementation Decisions-:

In order to reduce the duplicate error notifications to a node, we attach a reference list to each ROUTE ERROR. The node detecting a link failure becomes the root node. It initializes the reference list to be its notification list; each child only sends cache update notifications to nodes not on this list and updates this list by adding nodes on its own notification list. When using the cache update algorithm, we also use a small list of broken links like a negative cache to prevent a node from being re-polluted by the in-flight stale routes. Its size is set to 5 and timeout is set to 2 s for all scenarios used in simulations. This component is not part of the algorithm, which does not use any ad hoc parameter.

4 Performance Evaluation-:

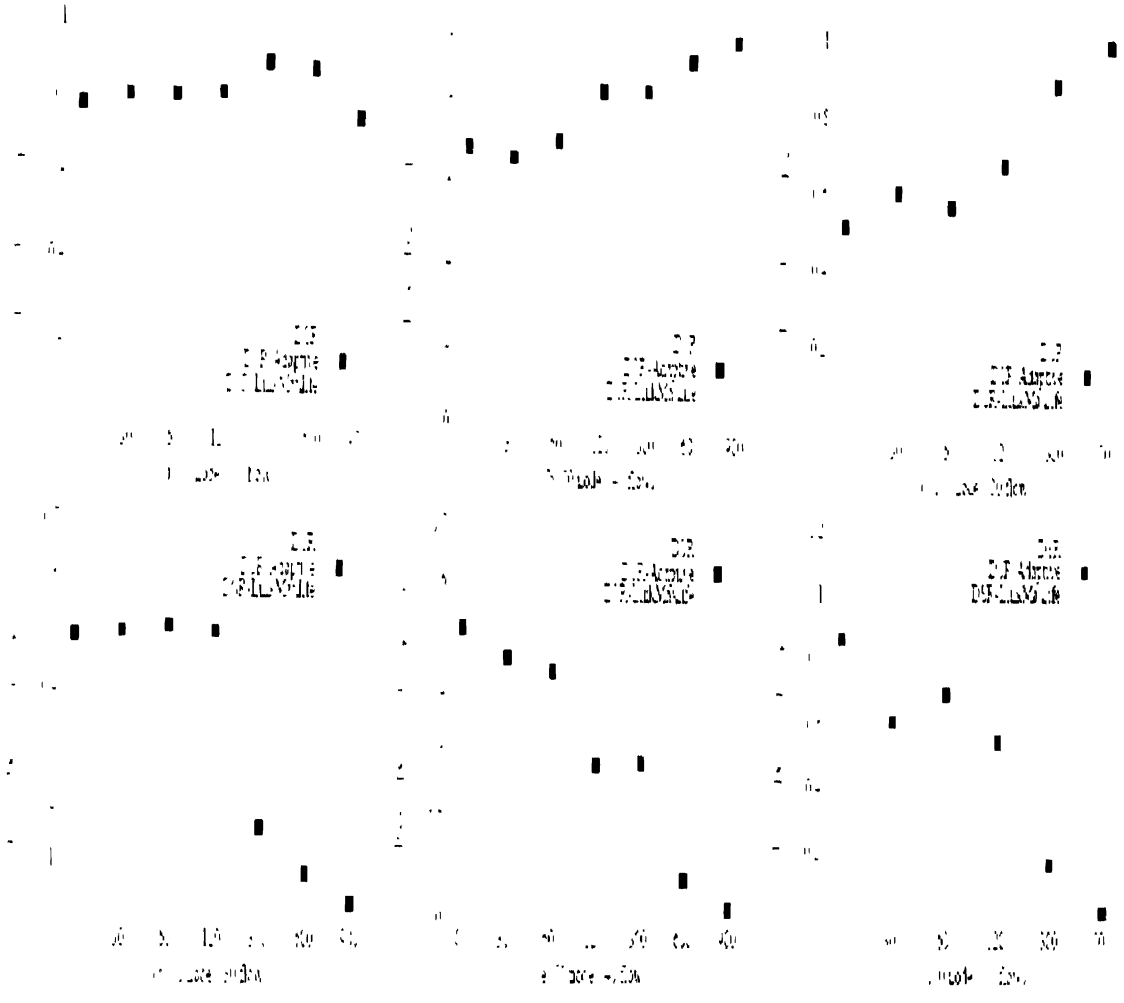


Figure 5 Packet Delivery Ratio and Packet Delivery Latency vs Mobility (Pause Time (s))

We used four metrics-:

1.Packet Delivery Ratio: the fraction of data packets sent by the source that are received by the destination.

2.Packet Delivery Latency: the average time taken by a data packet to travel from the source to the destination.

3.Normalized Routing Overhead: the ratio of the total number of routing packets transmitted to the total number of data packets received. For the cache update algorithm, routing packets include ROUTE ERRORS used for cache updates.

4.Good Cache Replies Received: the percentage of ROUTE REPLIES without broken links received by the source that originated from caches.

DSR-Adaptive outperforms DSR for all mobility scenarios, obtaining improvement of 13% for 50n-30f, 11% for 50n-40f, and 34% for 100n-30f at pause time 0 s. Such significant improvement demonstrates that our algorithm quickly removes stale routes. Since DSR with path caches has delayed awareness of mobility, more packets are dropped at intermediate nodes due to stale routes. As we can see, the improvement increases as mobility increases, showing the efficient adaptation of our algorithm to topology changes. The improvement is much higher for 100 node scenarios than for 50 node ones.

Fast cache updating is important for large networks, because more nodes will cache stale routes as network size increases.
These results show that proactive cache updating is more efficient than FIFO in invalidating stale information. DSR-Adaptive also outperforms Link-Maxlife, obtaining the maximum improvement of 16% for 50n-30f, 35% for 50n- 40f, and 20% for 100n-30f. For 50n-40f, the highest traffic load and higher node density scenarios, DSR-Adaptive significantly outperforms Link-Maxlife, achieving the highest im-provement. This is because as traffic load or node density increases, the adverse effects of stale routes become more sig- nificant: as traffic load increases, more traffic sources will use stale routes, resulting in more packet losses; as node density increases, more nodes will cache stale routes. These results show that Link-Maxlife cannot accurately predict timeouts.

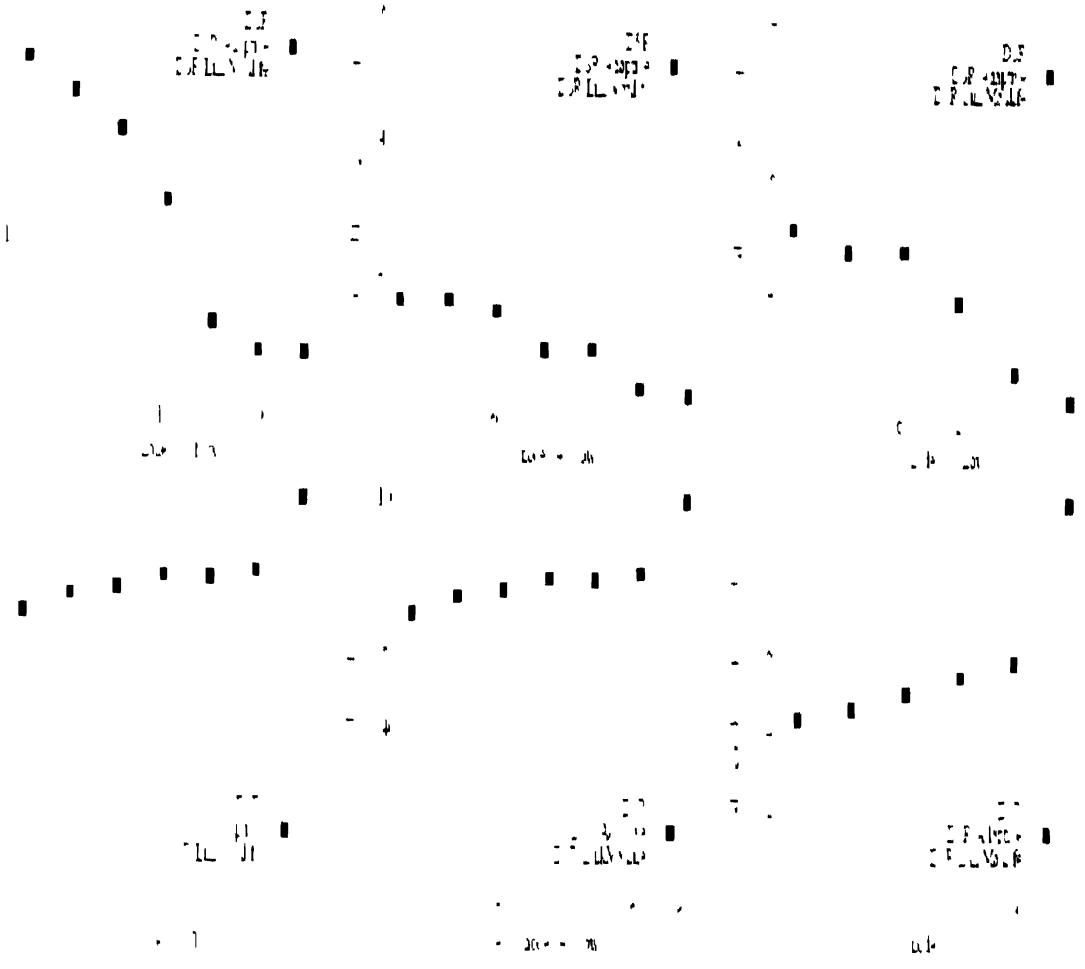


Figure 6: Normalized Routing Overhead and Good Cache Replies Received vs. Mobility (Pause Time (s))

Link-Maxlife performs better than DSR for high mobility scenarios, since it is able to aggressively expire links when links break more frequently under high mobility. This is consistent with the statistics of good cache replies received, as shown in Fig. 6 (d) and (f), where the cache performance of the former is better than that of the latter under high mobility. However, Link-Maxlife performs worse for other mobility scenarios, especially under high traffic load. This is because
when mobility is not high, path caches with FIFO remove stale routes faster than predicting timeouts, since high traffic load speeds up cache turnover.

For DSR and Link-Maxlife, there is an inconsistency be- tween good cache replies received and packet delivery ratio at pause time 0 s for 50n-40f and when mobility is not high for other scenarios. We attribute the following reason to this observation. Although more cache replies received are good in Link-Maxlife, FIFO replacement policy has evicted more stale routes when cached routes are picked up to be used.

Conclusions-:

We have presented a novel solution to the cache staleness issue of DSR. We define a cache table to collect and maintain the information necessary for cache updates. When a node detects a link failure, our distributed cache update algorithm proactively notifies all the reachable nodes whose caches con- tain the broken link to update their caches, using local information kept by each node and relying on cooperative update propagation. The algorithm does not use ad hoc parameters, thus making route caches fully adaptive to topology changes. Through detailed simulations, we show that the algorithm significantly improves packet delivery ratio and reduces packet delivery latency compared with DSR with path caches. It also considerably outperforms Link-Maxlife in packet delivery ratio. Our results lead to the following conclusions:

a.) Due to unpredictable topology changes, heuristics can not accurately predict timeouts. Proactive cache update ing is more efficient than adaptive timeout mechanisms in invalidating stale routing information.

b.) The effectiveness of predetermined choices of ad hoc parameters in caching strategies depends on scenarios. It is important to make route caches dynamically adapt to changing network characteristics.

Appendix: Pseudo Code

Procedures:

int Index(PATH *rte*, ID *id*);
compute the index of node *id* in route *rte*;

PATH subPath(PATH *rte*, ID *stnID*, ID *endID*);
compute from *rte* a subpath starting with index being *stnID* and ending with index *endID*;

boolean replyPathExist(vector <ReplyPath*>, *rpthRcvd*, ReplyPath *rpth_new*);
if *rpth_new* exists in *rpthRcvd* return true , else return false;

CacheEntry getFromCacheTable(PATH *rte*);
If some entry *e* in cacheTable satisfies *entry.rte* == *rte* return *e*, else return null;

boolean isFirstNode(PATH *rte*, ID *id*);
if *id* == *rte*.get(0) (first node) then return true, else return false;

boolean isLastNode(PATH *rte*, ID *id*);
if *id* == *rte*.get(*rte*.size() -1) (last node) then return true, else return false;

```
Input: PACKET p
1  if p is a RREP packet then
2    if netID = p.dest then
       e =
4    if e = null then
5      cacheTable = cacheTable ∪ {(p.srcRoute, p.srcRoute[0], p.srcRoute[p.srcRoute.length − 1], 0, 0)}
6    else
7      newRoute = subPath(p.srcRoute.Index(p.srcRoute.netID), Index(p.srcRoute.p.srcRoute[p.srcRoute.length − 1]));
8      reply_pair = (p.srcRoute.Index(p.srcRoute.netID) − 1, newRoute);
9      e = getFromCacheTable(newRoute);
10     if e = null then
11       cacheTable = cacheTable ∪ {(newRoute, p.srcRoute[0], p.srcRoute[p.srcRoute.length − 1], 0, reply_pair)}
12     else
13       if not replyPairExist(e.replyRecord, reply_pair) then e.replyRecord = e.replyRecord ∪ {reply_pair}
14 elseif p is a DATA packet then
15   e = getFromCacheTable(p.srcRoute);
16   if e ≠ null then
17     if e.DP = 1 then e.DP = 2
18   else
19     if netID = p.dest then cacheTable = cacheTable ∪ {(p.srcRoute, p.src, p.dest, 1, 0)}
20     else
21       for each entry e ∈ cacheTable do
22         if e.srcDest.src = p.src and e.srcDest.dest = p.dest and p.src = p.route[0] then
23           temp = subPath(p.srcRoute.Index(p.srcRoute.netID), Index(p.srcRoute.p.srcRoute[p.srcRoute.length − 1]));
24           if temp = e.route and e.DP = 0 then e.route = p.srcRoute; e.DP = 1; is_completed = TRUE
25         for each entry r ∈ e.replyRecord do
26           temp = subPath(p.srcRoute.Index(p.srcRoute.netID) − 1, Index(p.srcRoute.p.srcRoute[p.srcRoute.length − 1]));
27           if (r.nodeNotified ∪ r.subrouteSent) = temp then e.replyRecord = e.replyRecord \ {r};
28       if not is_completed and p.src = p.route[0] then cacheTable = cacheTable ∪ {(p.srcRoute, p.src, p.dest, 1, 0)}
```

Algorithm: *cacheUpdate*

```
Input: ID from, ID to, PACKET p, boolean detect_by_me, boolean continue_to_notify
/* If p is a ROUTE ERROR and p.src = from and netID = tellID, then continue_to_notify is set TRUE. */
Output: vector <NotifyEntry*> notifyList
 1 for each entry e ∈ cacheTable do
 2  if link (from,to) ∈ e.route then has_broken_link := TRUE; direction := forward
 3  elseif link (to,from) ∈ e.route then has_broken_link := TRUE; direction := backward
 4  else has_broken_link := FALSE;
 5  if has_broken_link then
 6   position = Index(e.route,from);
 7   if detect_by_me then
 8    if direction = forward then
 9     if (e.DP = 1 or e.DP = 2) and (not isFirstNode(e.route.netID)) then
10      notifyList = notifyList ∪ {(e.route[position-1], (netID, e.route[position-1]))}.
11     if e.DP = 2 or (e.DP = 1 and (not (p is a DATA packet and (p.srcRoute = e.route)))) then
12     routeToUse = ∅;
13      for each node n ∈ {e.route[position+1], ..., e.route[e.route.length-1]} do
14       Try to find a shortest route in cacheTable from netID to n
15       if such route is found then
16        foundRoute := the found route;
17        if routeToUse = ∅ or |foundRoute| < |routeToUse| then routeToUse := foundRoute; tellID := n
18      if routeToUse ≠ ∅ then notifyList := notifyList ∪ {(tellID, routeToUse)}
19    elseif direction = backward then
20     if not isLastNode(e.route.netID) then
21      notifyList := notifyList ∪ {(e.route[position+1], (netID, e.route[position+1]))}
22     routeToUse = ∅;
23     for each node n ∈ {e.route[position-1], ..., e.route[0]} do
24      Try to find a shortest route in cacheTable from netID to n
25      if such route is found then
26       foundRoute := the found route;
27       if routeToUse = ∅ or |foundRoute| < |routeToUse| then routeToUse := foundRoute; tellID := n
28     if routeToUse ≠ ∅ then notifyList := notifyList ∪ {(tellID, routeToUse)}
29   else /* The current node receives a notification from another node. */
```

```
27        if routeToUse = ∅ or |foundRoute| < |routeToUse| then routeToUse := foundRoute; tellID := n
28      if routeToUse ≠ ∅ then notifyList := notifyList ∪ {(tellID, routeToUse)}
29    else /* The current node receives a notification from another node. */
30      index := Index(e.route, netID);
31      if direction = forward and index < position and (not isFirstNode(e.route, netID)) then
32        notifyList := notifyList ∪ {(e.route|index − 1], (netID||e.route|index − 1])}
33      if direction = backward and index > position and (not isLastNode(e.route, netID)) then
34        notifyList := notifyList ∪ {(e.route|index + 1], (netID||e.route|index + 1])}
35      if (e.DP = 1 or e.DP = 2) and ((direction = forward and index > position) or
          (direction = backward and index < position)) then
36        if continue_to_notify then
37          if (direction = forward and netID = to and (not isLastNode(e.route, netID))) or
              (direction = backward and isFirstNode(e.route, netID) and (not netID = to)) then
38            notifyList := notifyList ∪ {(e.route|index + 1], (netID||e.route|index + 1])}
39          if (direction = forward and isLastNode(e.route, netID) and (not netID = to)) or
              (direction = backward and netID = to and (not isFirstNode(e.route, netID))) then
40            notifyList := notifyList ∪ {(e.route|index − 1], (netID||e.route|index − 1])}
41        if not (netID = to or (direction = forward and isLastNode(e.route, netID)) or
              (direction = backward and isFirstNode(e.route, netID))) then
42          notifyList := notifyList ∪ {(e.route|index + 1], (netID||e.route|index + 1])}
43          notifyList := notifyList ∪ {(e.route|index − 1], (netID||e.route|index − 1
44    for each entry r ∈ e.replyRecord do
45      if link (from, to) ∈ e.replyRecord.subrouteSent or link (to, from) ∈ e.replyRecord.subrouteSent then
46        tellID := e.replyRecord.nodeNotified;
47        notifyList := notifyList ∪ {(tellID, (netID||tellID))};
48        e.replyRecord := e.replyRecord \ {r};
49    cacheTable := cacheTable \ {e};
50  else /* The route in the table entry does not contain a broken link. */
51    for each entry r ∈ e.replyRecord do
52      if r.nodeNotified = to and netID = from then /* A broken link is detected by a ROUTE REPLY.
53        e.replyRecord := e.replyRecord \ {r};
54  for each entry n ∈ notifyList do
55    if (p is a ROUTE ERROR and n.tellID = p.src) or
        (n.routeToUse is a sub-route of another entry's routeToUse) or
        (entry m ∈ notifyList and n.tellID = m.tellID and |n.routeToUse| ≥ |m.routeToUse|) then
56      notifyList := notifyList \ {n};
57  return notifyList;
```

References:

[1] ***David F. Bantz and Frederic J. Bauchot.*** Wireless LAN Design Alternatives. IEEE Network, 8(2):43--53, March/April 1994.

[2] ***Vaduvur Bharghavan, Alan Demers, Scott Shenker, and Lixia Zhang.*** MACAW: A Media Access Protocol for Wireless LAN's. In Proceedings of the ACM SIGCOMM '94 Conference, pages 212-- 25, August 1994.

[3] ***Robert T. Braden, editor.*** Requirements for Internet Hosts--- Communication Layers. RFC 1122, October 1989.

[4] ***Scott Bradner.*** Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997.

[5] ***Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva.*** A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking, pages 85--97, October 1998.

[6] ***David D. Clark.*** The Design Philosophy of the DARPA Internet Protocols. In Proceedings of the ACM SIGCOMM '88 Conference, pages 106--114, August 1988.

[7] ***Stephen E. Deering and Robert M. Hinden.*** Internet Protocol Version 6 (IPv6) Specification. RFC 2460, December 1998.

[8] ***Ralph Droms.*** Dynamic Host Configuration Protocol. RFC 2131, March 1997.

[9] ***The FreeBSD Project.*** Project web page available at http://www.freebsd.org/.